Type inference in Kotlin

Svetlana Isakova

JetBRAINS

Programming languages

statically-typed

Java
C#
Scala
Kotlin

dynamically-typed

Groovy
Python
Ruby
JavaScript

Statically-typed Java

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
System.out.println(map.keySet()); // [1, 2]
map.keys();          // Compilation error
```

Dynamically-typed Groovy

```
def map = [1: 'one', 2: 'two']
map.keys()          // runtime exception: missing method
```

Statically-typed Kotlin

```
val map = mapOf(1 to "one", 2 to "two")
map.keys()          // Compilation error
```

Basic definitions

Type

describes a set of values

Examples: Int, String, List<String>

```
"abc".length()                    Int
3                                 Int
```

User types

```
class Foo { /*...*/ }

new Foo()
```

Foo

Is the set of types finite?

Theoretically it's infinite.

```
class List<T> { /*...*/ }
```

```
listOf("a", "bc")
```
List<String>

```
listOf(setOf(1, 2, 3))
```
List<List<Int>>

Subtype

$S \prec T$ S - is a subtype of T

Type is a set of values

Subtype is a subset of values

Examples: Int $\prec$ Any

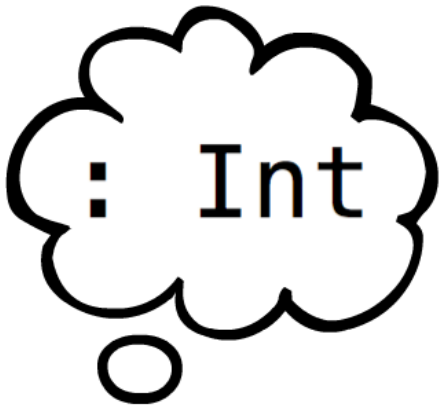
String $\prec$ String

ArrayList<String> $\prec$ List<String>

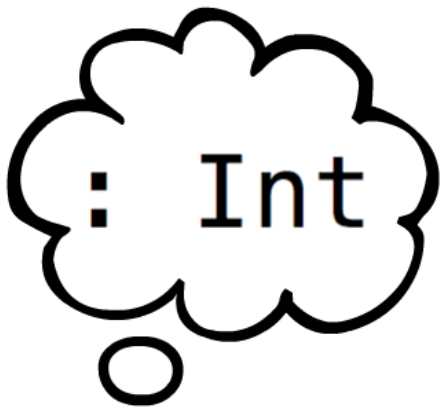
Type inference

- declarations
- type arguments
- expressions

Declaration type inference




```
val x = 42
```




```
fun square(y: Int) = y * y
```

Type arguments inference

```
fun listOf<T>(vararg values: T): List<T>
```

```
listOf("a", "bc")
```

T = String

Expression type inference

```
"abc"
```
String

```
"abc".substring(2)
```
String

Type inference of function invocation. 1. Simple function

Function invocation

```
fun foo(s: String): Int
```

```
foo("abc")
```

Int

String ≺ String

```
foo(1)
```

Int ≺ String

Function invocation (2)

```
fun foo(i: Int, a: Any): String


foo(1, "")
```

String

$$\begin{cases} \text{Int} \prec \text{Int} \\ \text{String} \prec \text{Any} \end{cases}$$

Type inference of function invocation (1)

```
fun foo(p1: P1, …  pn: Pn): R

foo(a1, …  an)                                  R
```

Find argument types $a_1: A_1 \quad \ldots \quad a_n: A_n$

Write down the constraint system;
check all the constraints

$$\begin{cases} A_1 \prec P_1 \\ \quad \ldots \\ A_n \prec P_n \end{cases}$$

Type inference of function invocation. 2. Overloaded functions

Overloaded functions

① `fun foo(i: Int)`
② `fun foo(i: Int, s: String)`
③ `fun foo(i: Int, a: Any)`

```
foo(1, "")    ②
```

Overloaded functions (2)

```
1 fun foo(i: Int, a: Any)
2 fun foo(a: Any, s: String)
```

```
foo(1, "")
```

overload resolution ambiguity

Type inference of function invocation (2)

`foo(a1, … an)`

Find all the functions named "foo"

For each one check
if it can be invoked

Find the most
specific one

Type inference of function invocation. 3. Generic functions

Generic functions

```
fun <T> foo(t: T): String
```

```
foo(1)
```

String

Int ≺ T | T = Int

Generic functions (2)

```
fun <T> foo(t1: T, t2: T): T
```

```
foo(1, 1.0)
```

Number

$$\begin{cases} \text{Int} \prec T \\ \text{Double} \prec T \end{cases}$$

T = Number

Type inference of function invocation (3)

```
fun foo<Ti,…Tj>(p1: P1<Ti>, …  pn: Pn<Tj>): R

foo(a1, …  an)
```

Find argument types

$a_1 : A_1 \ \ldots \ a_n : A_n$

Write down the constraint system;
check if the system has a solution

$$\begin{cases} A_1 \prec P_1\langle T_i \rangle \\ \ldots \\ A_n \prec P_n\langle T_j \rangle \end{cases}$$

$T_i = ? \ \ldots \ T_j = ?$

Reduction of constraint system

Constraint system reduction

```
fun <T> foo(list: List<T>): T
```

```
val list: List<Int>
foo(list)
```

$$\text{List<Int>} \prec \text{List<T>} \Rightarrow$$

$$\text{Int} \prec \text{T}$$

or

$$\text{Int} = \text{T}$$

(Im)mutable lists

correct:

ImmutableList<Int> ≺ ImmutableList<Any> 

incorrect:

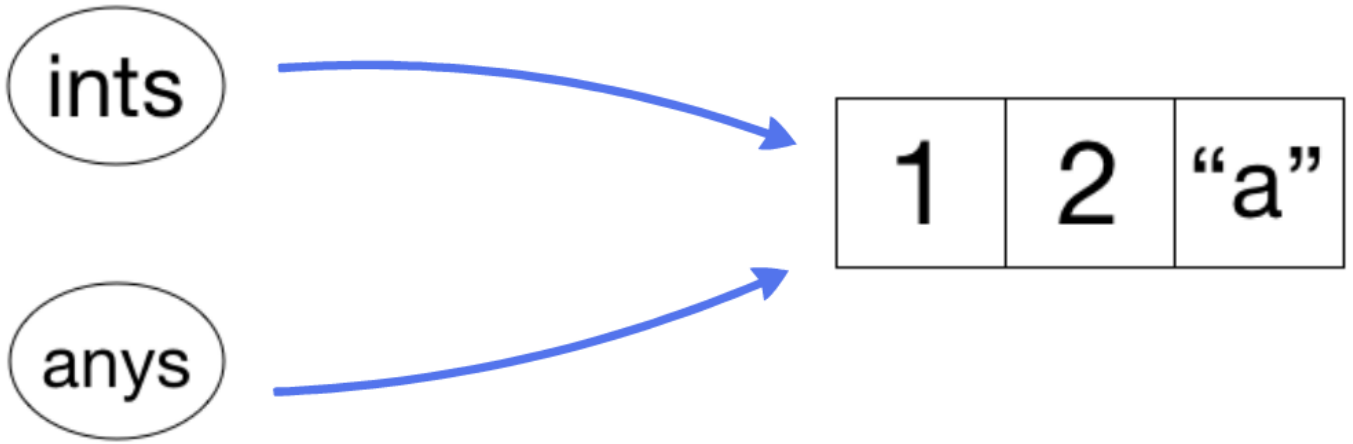
MutableList<Int> ≺ MutableList<Any> 

Why?

Imagine List<Int> was a subtype of List<Any>

```
val ints: List<Int> = listOf(1, 2)
val anys: List<Any> = ints
anys.add("a")
```

Now the list "ints" contains a string!



List in Kotlin

```kotlin
// read-only
interface List<out E> {
    fun get(index: Int): E
    // ...
}

interface MutableList<E> {
    fun add(e: E)
    // ...
}
```

Constraint reduction

$$\text{List}\langle T\rangle \prec \text{List}\langle R\rangle$$

$$\Rightarrow \quad T \prec R$$

$$\text{MutableList}\langle T\rangle \prec \text{MutableList}\langle R\rangle$$

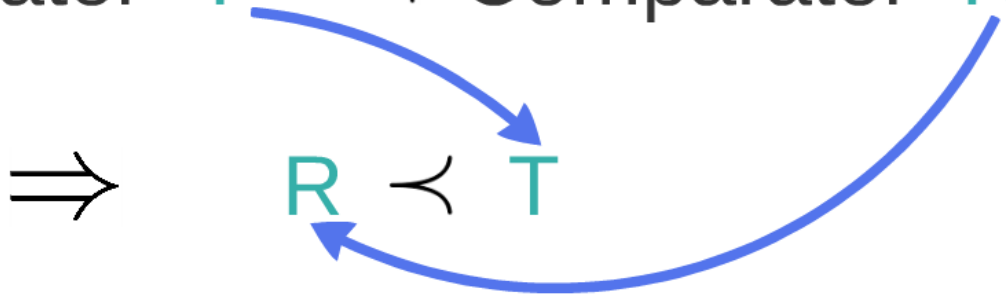
$$\Rightarrow \quad T = R$$

Constraint reduction (2)*

```
interface Comparator<in T> {
    fun compare(o1: T, o2: T): Int
}
```

Comparator<Any> ≺ Comparator<Int>

Comparator<T> ≺ Comparator<R>

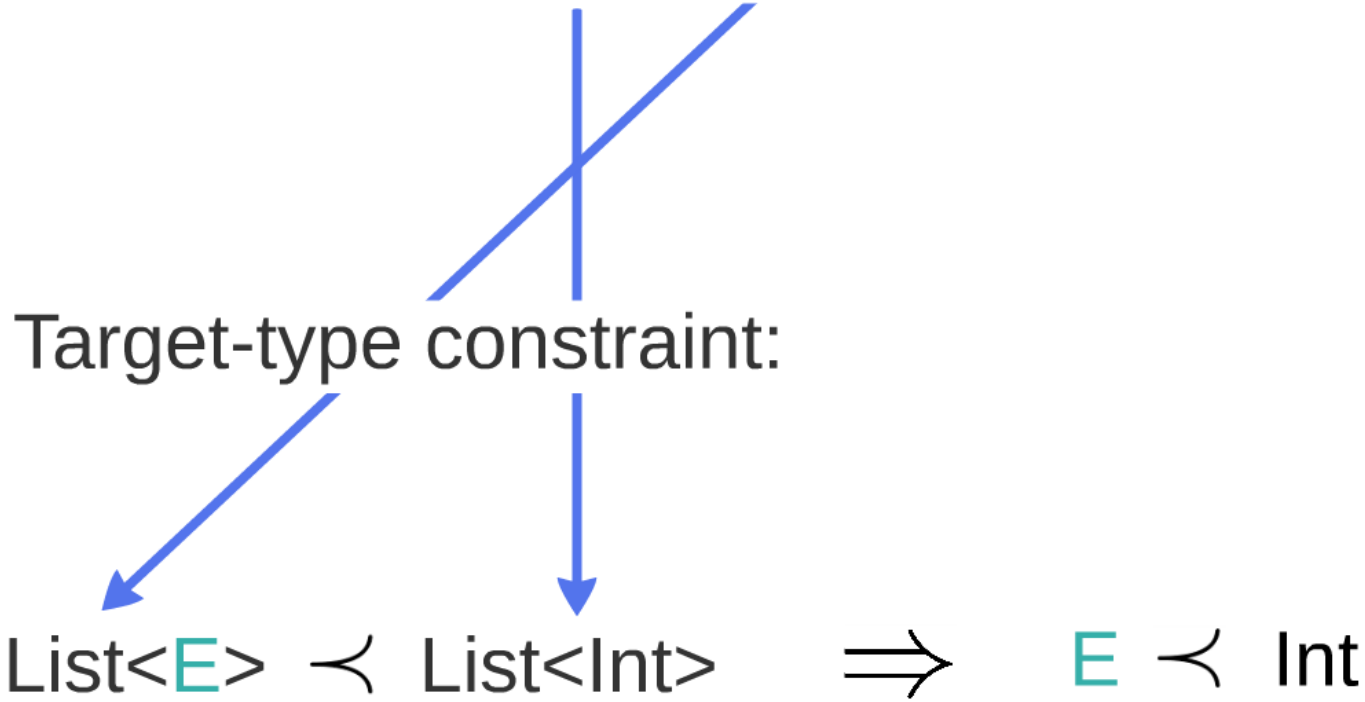


Type inference of function invocation. 4. Target type

'Target' type

```
fun emptyList<E>(): List<E>

val list: List<Int> = emptyList()
```



Target-type for argument

```
fun emptyList<E>(): List<E>
fun foo(list: List<Int>)

foo(emptyList())
```

* That's how Java 7 works

If target-type is not used*

`emptyList()` → List<Object>

{ ∅ | E = Object

List<Object> → List<Int> ✗

Why not just use target-type

```
bar(foo(emptyList()))
```

```
1 fun bar(/*...*/)
2 fun bar(/*...*/)
3 fun bar(/*...*/)
```

```
1 fun foo(list: List<Int>)
2 fun foo(set: Set<Int>)
3 fun foo(collection: Collection<Int>)
```

For each overloaded function 'bar' we analyze 'foo(emptyList())' with a corresponding target-type.

How many times 'emptyList()' is analyzed?

3 * 3 = 9

Argument type...

...is found due to unknown type variables

```
fun emptyList<E>(): List<E>
fun foo(list: List<Int>)

foo(emptyList())
```

`emptyList()` → `List<E>`

$\{ \varnothing$ | `E` = ?

`List<E>` $\prec$ `List<Int>` | `E` = `Int` ✓

Type inference of function invocation (4)

```
fun foo<Ti,…Tj>(p1: P1<Ti>, …  pn: Pn<Tj>): R

  foo(a1, …  an)
```

Argument types are found
due to unknown type variables

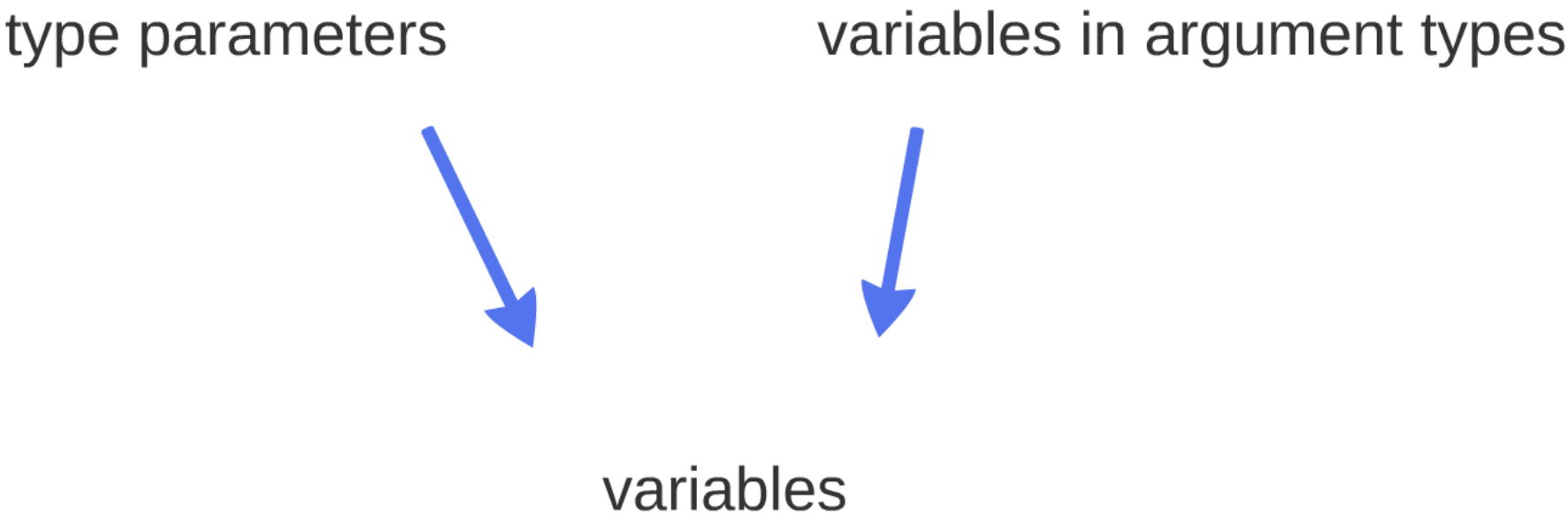
$a_1 : A_1\langle E_k \rangle \ \ldots \ a_n : A_n\langle E_l \rangle$

Write down the constraint system;
check if the system has a solution

$$\begin{cases} A_1\langle E_k \rangle \prec P_1\langle T_i \rangle \\ \ldots \\ A_n\langle E_l \rangle \prec P_n\langle T_j \rangle \end{cases}$$

$$T_i = ? \ \ldots \ T_j = ?$$
$$E_k = ? \ \ldots \ E_l = ?$$

Variables in a constraint system



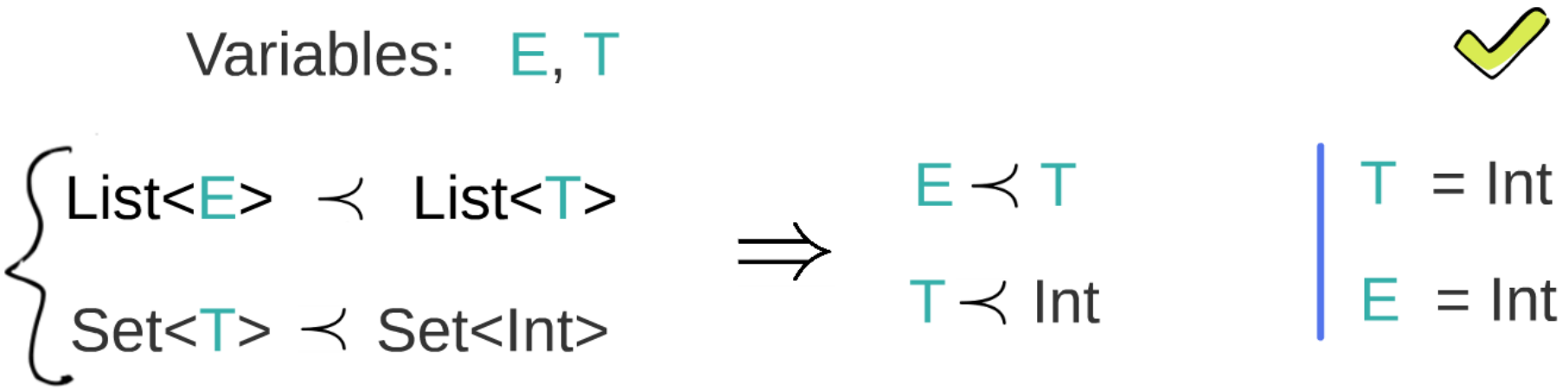
Common system for function and arguments

```
fun foo<T>(list: List<T>): Set<T>

fun emptyList<E>(): List<E>

val set: Set<Int> = foo(emptyList())
```

Variables: E, T



Real-life example

Real-life example: 'toCollection()'

```
fun <E> newHashSet(): HashSet<E>
val list: List<Int>
list.toCollection(newHashSet())
```

wanted:
HashSet<Int>

If the return type of 'toCollection' was Collection<T>, the result type would be Collection<Int>, not HashSet<Int>.

System for 'toCollection()'

```
fun <E> newHashSet(): HashSet<E>
val list: List<Int>
list.toCollection(newHashSet())
```

```
fun <T, C: MutableCollection<T>>
    Collection<T>.toCollection(c: C): C
```

Variables: E, T, C

List<Int> ≺ Collection<T> — receiver constraint

HashSet<E> ≺ C — argument constraint

C ≺ MutableCollection<T> — upper bound constraint

Solution for 'toCollection'

```
list.toCollection(newHashSet())
```

Variables: E, T, C

List<Int> ≺ Collection<T>

⇒ Collection<Int> ≺ Collection<T>

⇒ Int ≺ T

HashSet<E> ≺ C

C ≺ MutableCollection<T>

⇒ HashSet<E> ≺ MutableCollection<T>

⇒ E ≺ T

T = Int E = Int C = HashSet<Int>

'toCollection()'!

```
fun <E> newHashSet(): HashSet<E>
val list: List<Int>
list.toCollection(newHashSet())
```

HashSet<Int>

```
fun <T, C: MutableCollection<T>>
    Collection<T>.toCollection(c: C): C
```

Variables: E, T, C

T = Int

E = Int

C = HashSet<Int>

Type inference of function invocation. 5. Lambdas

Lambdas

```
fun <T> filter(
    list: List<T>,
    predicate: (T) -> Boolean
): List<T>
```

```
val list: List<Int>

filter(list, { i: Int -> i < 42 })
```

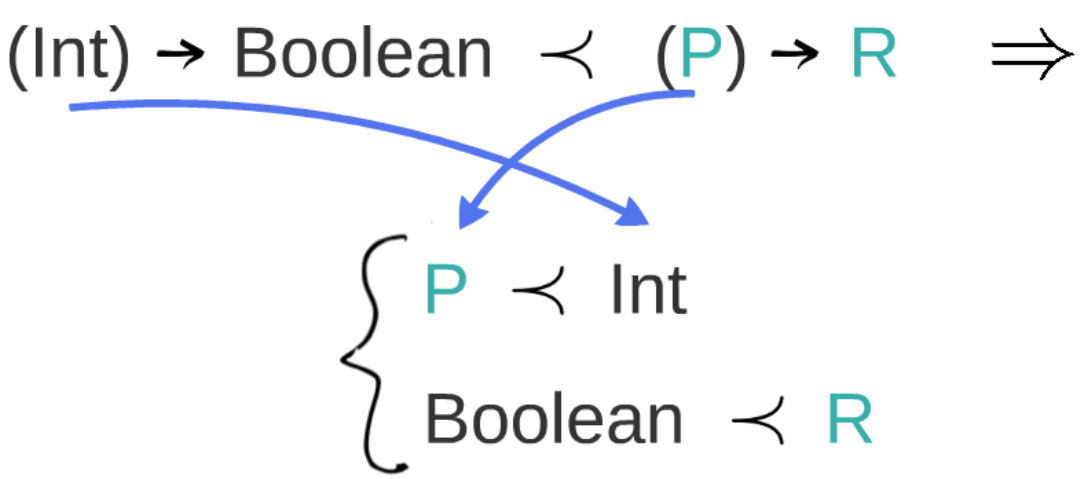
From the constraint on 1st argument (list): $\text{Int} \prec T$

From the constraint on 2nd argument (lambda): $T \prec \text{Int}$

$T = \text{Int}$

Kotlin functions

```
interface Function1<in P1, out R> {
    fun invoke(p1: P1): R
}
```



Lambdas, solution

```
fun <T> filter(
    list: List<T>,
    predicate: (T) -> Boolean
): List<T>

val list: List<Int>

filter(list, { i: Int -> i < 42 })
```

$$\begin{cases} \text{List<Int>} \prec \text{List<T>} & \Rightarrow \quad \text{Int} \prec \text{T} \\ \text{(Int)} \to \text{Boolean} \prec \text{(T)} \to \text{Boolean} & \Rightarrow \quad \text{T} \prec \text{Int} \end{cases}$$

$$\text{T} = \text{Int}$$

Implicitly-typed lambdas

```
fun <T> filter(
    list: List<T>,
    predicate: (T) -> Boolean
): List<T>

val list: List<Int>
```

implicitly-typed lambda

```
filter(list, { i -> i < 42 })
```

⇓

T = Int

- When the lambdas body is analyzed?..
- The body of an implicitly-typed lambda is analyzed after the constraint system is solved and the variables are fixed!

Type inference of function invocation (2)

`foo(a1, … an)`

Find all the functions named "foo"

For each one check
if it can be invoked

Find the most
specific one

Type inference of function invocation (4)

```
fun foo<Ti,…Tj>(p1: P1<Ti>, …  pn: Pn<Tj>): R

  foo(a1, …  an)
```

Argument types are found
due to unknown type variables

$a_1 : A_1\langle E_k \rangle \ \ldots \ a_n : A_n\langle E_l \rangle$

Write down the constraint system;
check if the system has a solution

$$\begin{cases} A_1\langle E_k \rangle \prec P_1\langle T_i \rangle \\ \ldots \\ A_n\langle E_l \rangle \prec P_n\langle T_j \rangle \end{cases}$$

$$T_i = ? \ \ldots \ T_j = ?$$
$$E_k = ? \ \ldots \ E_l = ?$$

Type inference of function invocation (5)

Find argument types

$a_1 : A_1\langle E_k\rangle \ \ldots \ a_n : A_n\langle E_l\rangle$

Collect all functions with given name

For each function compose the constraint system

$$\begin{cases} A_1\langle E_k\rangle \prec P_1\langle T_i\rangle \\ \ldots \\ A_n\langle E_l\rangle \prec P_n\langle T_j\rangle \end{cases}$$

(the system has contradiction => function is unapplicable)

Choose the most specific one

In a loop, while system isn't solved:

- simplify the constraint system

$$\begin{cases} A_1\langle E_k\rangle \prec P_1\langle T_i\rangle \\ \ldots \\ A_n\langle E_l\rangle \prec P_n\langle T_j\rangle \end{cases} \Rightarrow \ldots$$

- fix variables

$T_i = \ldots$
$E_k = \ldots$

- analyze implicitly-typed lambdas

```
{ i -> … }
```

'map'

```
fun <T, R> map(
    list: List<T>,
    transform: (T) -> R
): List<R>

val list: List<Int>

map(list, { i -> "$i" })
```

List<String>

Int ≺ T

String ≺ R

Lambda adds a new constraint to a system!

'map' * 2

```
fun <T, R> List<T>.map(
        transform: (T) -> R
): List<R>

val list: List<List<Int>>

list.map { l -> l.map { i -> "$i" } }

list.map { it.map { i -> "$i" } }
```

* "map" is an extension-function

List<List<String>>

List<String>**

**Actually, not List<String>, but List<R'>

Int ≺ T'

String ≺ R'

Lambda adds new variables and constraints to the system!

Type inference of function invocation (5)

Find argument types

$a_1 : A_1\langle E_k\rangle \ \ldots \ a_n : A_n\langle E_l\rangle$

Collect all functions with given name

For each function compose the constraint system

$$\begin{cases} A_1\langle E_k\rangle \prec P_1\langle T_i\rangle \\ \ldots \\ A_n\langle E_l\rangle \prec P_n\langle T_j\rangle \end{cases}$$

(the system has contradiction => function is unapplicable)

Choose the most specific one

In a loop, while system isn't solved:

- simplify the constraint system

$$\begin{cases} A_1\langle E_k\rangle \prec P_1\langle T_i\rangle \\ \ldots \\ A_n\langle E_l\rangle \prec P_n\langle T_j\rangle \end{cases} \Rightarrow \ldots$$

- fix variables

$T_i = \ldots$
$E_k = \ldots$

- analyze implicitly-typed lambdas

```
{ i -> … }
```

Thank you!